

For Reference

NOT TO BE TAKEN FROM THIS ROOM

Ex LIBRIS
UNIVERSITATIS
ALBERTAENSIS





Digitized by the Internet Archive
in 2023 with funding from
University of Alberta Library

<https://archive.org/details/Shapiro1972>

THE UNIVERSITY OF ALBERTA

COMBINING INDEPENDENTLY-COMPILED
ALGOL 68 PROGRAMS

BY



VICTOR SHAPIRO

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
AND RESEARCH IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL, 1972

THE UNIVERSITY OF ALBERTA
FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read,
and recommend to the Faculty of Graduate Studies and
Research for acceptance, a thesis entitled COMBINING
INDEPENDENTLY-COMPILED ALGOL 68 PROGRAMS submitted by
Victor Shapiro in partial fulfillment of the
requirements for the degree of Master of Science.

ABSTRACT

The purpose of this thesis is to investigate the processes involved when combining independently-compiled ALGOL 68 programs. The difficulties which exist due to certain features of ALGOL 68 are explained. A possible solution to these difficulties is then described in detail.

ACKNOWLEDGEMENTS

I wish to express my appreciation to my supervisor Professor B.J. Mailloux for his patience, advice, and guidance throughout the preparation of this research.

The financial assistance supplied through a scholarship from the National Research Council of Canada is gratefully acknowledged.

TABLE OF CONTENTS

	Page
Chapter 1: Introduction	1
Chapter 2: The Linking Programs	3
2.1 Library Builder	4
2.2 Linkage Editor	6
2.3 Loader	8
Chapter 3: The Requirements Of ALGOL 68	13
3.1 Modes	14
3.2 Compilation Time Requirements	16
3.3 Execution Time Requirements	18
Chapter 4: Additions To ALGOL 68	22
4.1 Details Of The Additions	22
4.2 Facilities Provided	25
Chapter 5: New Tasks For The Compiler And Linking Programs	30
5.1 The Compiler	30
5.2 The Linking Programs	41
Chapter 6: Algorithms	47
6.1 Equivalence Of Modes	47
6.2 Specifics Of The Linking Programs	58

Chapter 7:	Alternative Methods	63
Chapter 8:	Summation	69
References		71

LIST OF FIGURES

	Page
Figure 1: Format of a Mode CSECT	31
Figure 2: Sample Mode List and Tag Table	37
Figure 3: Format of a Section	38

Chapter 1

Introduction

The need to be able to design large computer programs in terms of smaller, more easily managed, subprograms and to provide accessible libraries of frequently used routines is well known. This paper discusses the processes involved when combining ALGOL 68 subprograms to perform one complete function.

ALGOL 68 is defined by the "Report On The Algorithmic Language ALGOL 68". It was intended that this language be a successor to ALGOL 60. A complete implementation of ALGOL 68 is being attempted at the University of Alberta.

There are various features of ALGOL 68 which provide programmers with very powerful and general facilities, different from other languages. One such facility is the variety of available data types (which are described in detail below). However such facilities necessitate nonstandard tasks to be performed when implementing the language. The features of ALGOL 68 which influence the tasks involved when combining ALGOL 68 subprograms will be

discussed.

There are several types of programs which are commonly used to do the standard job of combining subprograms. These are loaders, linkage-editors, and library builders. The term 'linking programs' will be used in reference to these three programs. A brief description of their tasks will be given in order to provide some understanding of the standard linking functions which are done when combining subprograms. This description is also necessary in order to understand the implementation of several nonstandard tasks which the linking programs must perform in order to combine ALGOL 68 subprograms.

Several solutions to the problems involving the combining of independently compiled ALGOL 68 programs will be discussed. The solution used in this implementation of the language will be outlined in detail. The tasks required of programmers who wish to combine ALGOL 68 subprograms, as well as the job of the ALGOL 68 compiler will be described. Details of the nonstandard jobs which the linking programs will need to perform are also given.

Chapter 2

The Linking Programs

The technique of modular programming is widely used; it involves the segmentation of programs into logical units called modules. The reasons for such segmentation are:

- 1) The coding and testing of individual modules can be done in parallel;
- 2) Coding can be reduced by having widely-used modules in libraries which are available to many programs (thus, also reducing the amount of testing time);
- 3) Modifications can be made to a single module without having to retranslate the whole program;
- 4) Different programming languages can be used for the different modules.

Because these modules are translated independently, it is necessary to be able to combine them into a complete program.

There are two basic tasks involved in the combining of modules to form a complete program for execution. The first is the resolving of symbolic references between modules. Consider two modules A and B. If the identifier SAMPLE is defined in A, and thus its value exists in A, then

B may contain a symbolic reference to SAMPLE. When B is combined with A, this reference must be resolved by placing the address of the value of SAMPLE where the reference, to SAMPLE, was made in B.

References may also exist within a module. If there is a reference to SAMPLE in A then the relative address of SAMPLE is placed, by the language translator, where the reference is made. A relative address is the displacement of a location from some origin. When modules are loaded, all relative addresses must be changed to actual memory addresses. This process, called relocation, is the second basic task when combining modules.

The three programs involved in this combining process are the loader, linkage editor and library builder. A brief description of the functions of, and relationships amongst, these linking programs will be given in this chapter. The details of this description pertain to the programs which are used in the MTS operating system at the University of Alberta Computing Center, but the generalities apply to all such programs.

2.1 Library Builder

Libraries form an important part in the application of the technique of modular programming. They are the

structures used for storing commonly used modules so they can be easily accessed. A library builder is simply a program which builds libraries. The following is a brief description of the form of libraries used here and how they are constructed by library builders.

There are two basic parts of a library. The first is the part in which all the modules in the library reside. The second is the directory, which contains names by which each module in the library may be referenced and also the location of each module. The library is designed to speed up the locating of modules; for, instead of reading the whole of each module to determine whether it is required, a simple search through the directory is sufficient. If the name of a required module is found, then the module can be read from the library starting at the location specified in the directory. Note that if the names in the directory are ordered and a good search method (such as a binary search) is employed, then the process of searching for desired modules in a library can be very fast.

A library builder receives as input a set of modules from which it forms a library. The input modules may be separate, or already in some library. The first task of a library builder is to determine the size of the directory. This is done by reading through all the input modules to determine the names by which the modules may be referenced.

Then the modules are put into the library one at a time. As each module is put into the library, each name by which it can be referenced is put into the directory along with the location of the module. When all the modules are in the library, the names in the directory can be sorted. The library is then complete.

2.2 Linkage Editor

The basic function of a linkage editor (see [1],[2] and [3] for more detail) is the processing of an indeterminate number of input modules so as to form a single output module. This output module contains all the input modules as though they had all been translated together. A brief description of the input, output, and type of tasks done by the linkage editor follows.

The input to a linkage editor consists of a combination of object modules, load modules and linkage editor control statements. An object module is the output from a language translator and consists of one or more control sections (CSECTs). A control section is a unit of coding (instructions and data) in which the displacement of any position from another position is known and fixed. All the references between CSECTs in an object module are resolved by the language translator. It is the linkage editor's job to resolve all references between CSECTs of

different input modules and to produce a single module from them, with all the crcss references resolved. This output module is called a load module.

Object and load modules are of the same form. First are external symbol dictionary (ESD) records. These records consist of various fields, one of which is the ESD 'type' which identifies the form (and content) of the remainder of the fields. ESD records define the various external symbols contained (defined) within the module and designate which symbols external to the module are referenced from within it. Next are text (TXT) records which contain the machine code (instructions and data) for the module. Following the TXT records are relocation dictionary (RLD) records. These contain the relocation information necessary to relocate any relative addresses in the code. Finally, an end (END) record designates the end of the module and may specify the entry point of the program (i.e., where instruction execution is to begin).

In order to form a load module from the input modules, the linkage editor must perform several tasks. It must construct new ESD records from the old ones. That is, if one module defines an external symbol and another module refers to that symbol, then the reference should be dropped from the final load module and a relative address replaces the symbolic reference. In any module the external symbols,

which refer to locations in the code from which relocation may be done, are assigned an identification number (ESID). This number, rather than an external symbol, is contained in the relocation information to designate positions in the code. When the ESD records are formed for the output load module their ESIDs must be renumbered (as ESIDs from different input modules will likely be conflicting) so as to be uniform and unique. When this is done the ESIDs must be renumbered in the RLD records to reflect the new set of ESD records. The output load module is constructed from the final ESD records with the resolved references removed, followed by the total text of all the input modules and the RLD records with their ESIDs properly renumbered, and finally the END record.

The linkage editor is used to speed up the process of loading modules. The loader is saved from having to resolve those references which the linkage editor resolves when it formed a load module. Thus, for production programs, use may be made of the linkage editor to combine the modules rather than having this done by the loader every time the program is to be executed. However, for non-production programs, use of the linkage editor is less useful.

2.3 Loader

The function of the loader (see [4] and [5] for more

detail) is to take an input of object modules, load modules, and control statements and to then put the text of the modules into the memory of the computer so that it can be executed. There are at least three types of loaders, distinguished by the jobs which they are capable of performing. The three basic jobs are: loading into memory, relocating of address constants, and resolving of cross-references between the loaded modules. A 'linking loader', the type which will be described, performs all three of these jobs. The other types of loaders are a 'binary loader' (a loader of this type simply places code into memory), and a 'relocating loader' (a loader of this type can do relocation as well as loading into memory).

As the loader reads each input module it places the external symbols from the ESD records into a table (ESD table) along with all the information defining their type and location. Another table is kept of the ESIDs of the external symbols of the module. This ESID table also contains the address in the ESD table of the corresponding external symbol for each ESID. After the ESD records are read, the TXT records are put into memory. Then the RLD records are read and the specified relocations are attempted.

Relocation is the process of putting into the loaded text the correct memory addresses in place of the relative

addresses. The RLD records contain the locations of the relative addresses and the locations of the points with respect to which the addresses are relative. The location of a relative address is specified by the ESID of an external symbol, which identifies a place in the program. The location with respect to which the address is relative is specified by the ESID of the external symbol identifying that point in the program. If the external symbols referenced by the ESIDs in an RLD record are defined (that is, the code which they reference is loaded) then the relocation can be done. The loader determines where the relative address is by taking the ESID which specifies the location of the relative address and searching for that ESID in the ESID table. When the ESID is found, the address of the external symbol in the ESD table is available. The information included with the external symbol in the ESD table contains the address at which it was loaded, and thus the location of the relative address is known. The address of the external symbol with respect to which the relocation is done is also found by use of the ESID table. The address with respect to which the relocation is done is added to the relative address, which is then replaced by the sum. All relocation is done in this fashion.

If the external symbol referring to the location with respect to which the relocation is done is not defined when the relocation is attempted then the relocation can not

yet be done. In this case the relocation information is placed into a table of 'forward references' and the relocation is attempted later when the necessary external symbols are defined. This table contains essentially the same information as the RLD records except that the ESIDs are replaced by the addresses in the ESD table of the corresponding external symbols. This is done because there is a new ESID table for each module (and old ESID tables are lost) since ESIDs are module-dependent and, thus, duplication of ESIDs may exist between modules. After each module is loaded the table of forward references is checked to see if any of them can be resolved. If, after all the available modules have been loaded, some relocations still need to be done then an error message is produced.

The loader does the same resolution of cross-references between modules as the linkage editor. However, the loader must resolve all external references in order for the loaded program to be able to execute properly. If any are not resolved when the normal input has been processed, then the loader checks for the symbols in two places, the system library and the low core symbol dictionary (a term used in MTS), which is a directory of resident system routines. If there are still unresolved references, then loading is stopped and an error message is produced. Otherwise the program is ready for execution.

The loader, while doing the above work, also determines the entry point of the program (which is the beginning of the first loaded CSECT unless otherwise specified on an END record or ENTRY type control statement). It also can produce a 'cross-reference table', consisting of all the external symbols which were referenced by the various modules, and the points from which the references were made. The loader can also produce a 'map' of the modules which were loaded, specifying the name and loaded address of each module. These functions essentially complete the loader's work.

This brief description of the standard jobs which are done by the linking programs is intended to aid in understanding the new jobs which will be assigned to these programs.

Chapter 3

The Requirements Of ALGOL 68

Having discussed the standard functions of the linking programs, we shall now examine those characteristics of ALGOL 68 which may necessitate additional tasks to be carried out by these programs. A main program may execute with an arbitrary number of procedures. If any of the procedures were compiled independently then difficulties arise. In order to compile an ALGOL 68 program ('program', in reference to ALGOL 68, will be used to mean a main program or procedure), the compiler needs to have certain information about each identifier (including those from programs which are independently compiled) which the program presently being compiled uses. The problem, then, is to supply the compiler with this necessary information. There are certain situations which require, at execution time, that the independently compiled programs which are being executed together appear as though they had been compiled together. The problem arises, therefore, of linking these programs together so as to remove certain compilation dependencies (such as different sets of mode numbers, as described below). The reasons that these dependencies must be removed, as well as the nature of the other problems

mentioned above, will be discussed in more detail in this chapter, and solutions will be presented.

3.1 Modes

The problems which were mentioned above all center around the modes (data types) used in the programs. ALGOL 68 has an unbounded number of modes, of which an ALGOL 68 program may use any (finite) subset. In ALGOL 68, a programmer can declare modes (see [6] for details on all ALGOL 68 syntax and terminology) which he wants to use. The primitive modes in ALGOL 68 are specified by the declarers:

int (integer number)
real (floating-point number)
bool (boolean value)
char (character)
format (for output or input)

Programmers may specify other modes in terms of these primitive ones by using the following affixes:

proc (procedure)
struct (structured value with selectors (names) for its fields)

union (For example, an expression of mode union (real , int) has a value of either mode real , or mode int , and this mode may change during program execution.)

ref (reference to , essentially, an address)
long (this refers to precision or range)

[], [], [], [], etc. (arrays)

Any compiler must somehow associate a mode with each identifier in a program. This association can be facilitated by the obvious device of assigning a number to each mode, and then, when an identifier is declared to be of some mode, the appropriate 'mode number' may be stored in the symbol table with the identifier.

Most languages have a small, fixed number of modes. For such languages the mode numbers used by their compilers can easily be the same from compilation to compilation. A fixed number of modes and a consistent (between compilations) set of mode numbers allows easy checking, at execution time, of the mode of the value of an identifier from an independently compiled program. This checking, although usually not done for most other languages, is sometimes necessary (as will be seen later) when executing ALGOL 68 programs.

In ALGOL 68, practical limits can be placed on the number of different modes specified using ref or long but no such limits exist for modes using the other affixes. Since the number of possible modes is unlimited, it is therefore impossible for an ALGOL 68 compiler to assign a unique mode number to each mode. The compiler assigns a mode number only to each mode used in the program being compiled. As

different programs may use different sets of modes, a compiler will not likely use the same correspondence between numbers and modes for every compilation.

The incompatibility of mode numbers from compilation to compilation needs to be overcome, due to requirements at both compilation and execution time, in order that independently compiled programs may be executed together. We will now consider in more detail the nature of this incompatibility.

3.2 Compilation Time Requirements

In dealing with modes, the first job a compiler has is to keep a description of each mode. This description (see 5.1 for more detail) is based upon the fact that any 'declarer' of a mode must be finite. These descriptions are kept in a 'mode list'. The next (or simultaneous) job the compiler has is associating each identifier, indication (an abbreviation for a declarer), and operator, with its mode. This can be done by using as mode number the displacement of the beginning of the mode description in the mode list, and storing these mode numbers in the symbol table with the appropriate identifiers.

The compiler then has the information necessary to handle statements (such as assignments) in the source program. Consider the assignment:

`x:=y`

Assume that `x` was declared to be of the mode specified by the declarer `ref amode` and `y` was declared to be of the mode specified by the declarer `bmode`. The compiler must check whether the mode-declarers `amode` and `bmode` are equivalent (see 6.1 for more details), i.e., whether they specify the same mode. Consider the following mode-declarations for `amode` and `bmode` :

```
mode amode = struct ( ref struct ( ref amode first, bool
                        second) first, bool second);
```

```
mode bmode = struct ( ref bmode first, bool second);
```

It is not too difficult to see that `amode` and `bmode` are indeed equivalent and that the assignation is valid.

So we see that the compiler, after it has a description of each mode used in a program, must renumber the mode descriptions to reflect any equivalences. After doing this, it is capable of handling the modes in the statements in the source program.

If the identifiers do not, a priori, have appropriate mode numbers, then the compiler must determine whether the modes of the identifiers may be coerced to be appropriate. (Coercions are certain implicit changes of modes, called type conversions in other languages. See [6] for more details.) If such coercions are possible, then the compiler must determine which changes of mode have to occur

and include code in the object program to compute the new values corresponding to the changed modes. An example of an assignation which involves coercion is $x:=y$, where x is declared to be of the mode specified by the declarer ref real and y is declared to be of that specified by int (henceforth the declarer will be used to stand for the mode). Here, the value of y must be coerced to the mode real before the assignment can be performed.

As can now be seen, the problem at compilation time is that the compiler needs to determine the mode of each identifier (including those from independently compiled programs) which has applied occurrences (an occurrence of an identifier other than its defining one, in a declaration) in the program being compiled. The compiler needs to have this information in order to check for equivalence of modes, and perform coercions between modes. How the compiler can be provided this information is described in the next chapter.

3.3 Execution Time Requirements

A problem of mode determination occurs at execution time when a program contains 'conformity relations'. A conformity relation (as described in [6]; note that syntax changes will not change the concepts described) is of the form:

$$x::=y$$

The modes of the values of x and y are, in general, indeterminable at compile time. For this form, the current value of y will be assigned to x if the mode of the current value of x (with at most certain allowed coercions), is the same as the mode of the current value of y (with at most certain allowed coercions). An obvious way to compare two modes is to compare their associated mode numbers. However, as we have seen, the mode numbers are compilation-dependent. Thus if x and y are, or contain, identifiers from different independently compiled programs, then comparison of their mode numbers will not be meaningful unless the mode numbers of the independently compiled programs have been made consistent.

Before independently compiled programs can be executed together, therefore, the modes in all these programs must be renumbered so that one uniform set of mode numbers is used by all. To facilitate this renumbering, and for later convenience (as described below), a single mode list is built which has a description of all the modes (with duplicates removed) used in these programs. Each new mode number is the displacement from the beginning of the composite mode list at which the mode description begins. Each occurrence of a mode number in the code must then be replaced by the appropriate new mode number.

This 'unified' system at execution time is useful

for 'garbage collection'. Garbage collection (see [7] and [8]) is the reshuffling of storage so as to recover, for usage by the program, storage which it can no longer access. Because of the dynamic storage allocation features (such as "heap" variables and "flex" arrays, whose storage is allocated during execution time) provided by ALGOL 68, it is virtually mandatory that a garbage collector be provided. If the region ('heap') in which this storage is allocated becomes full then a garbage collector is called in to reclaim any space which is no longer accessible to the program. The garbage collector needs to determine the structure of any value with which it is dealing, and the mode list provides this information. Without such a unified system, the garbage collector would need to determine, with each instance of a mode number, which mode list to use to find the correct mode description. This would require the use of additional information with each block of code, to designate the appropriate mode list. The garbage collector would, of course, also need access to all the mode lists of the different modules. Thus garbage collection is made simpler and the amount of storage needed for the program is decreased by this unified system.

Other routines, such as a general move routine, are also made simpler by this unified system. For instance, a general move routine may be employed to move all of, or

portions of, a value. A mode description is needed, to provide information about the structure of the value concerned. Thus complications are removed by using this unified system.

In summary, it is clear that it is necessary to remove compilation dependencies of modes by execution time. That is, there must exist a single mode list and uniform set of mode numbers for all the modules being executed together, just as if they had been compiled together rather than independently.

Those characteristics of ALGCL 68 which cause difficulties at compilation or execution time, when a main program and independently compiled procedures are to be executed together, have been discussed. Solutions to these difficulties have been presented. Implementation of these solutions is explained in detail in the following chapters.

Chapter 4

Additions To ALGOL 68

As was discussed in the previous chapter, an ALGOL 68 compiler must have information regarding each identifier (specifically, its mode) which any program being compiled will need in order to be able to execute. This information can be supplied to the compiler by the programmer if certain additions are made to ALGOL 68. Details of these additions as well as a description and evaluation of the facilities these additions allow programmers, in regard to executing together independently compiled programs, are given in this chapter. An alternative method for providing the compiler with the necessary information is described and evaluated in Chapter seven.

4.1 Details Of The Additions

A programmer must inform the compiler about those identifiers which have their defining occurrences (the occurrence of an identifier at which it is declared) in an independently compiled program, and those which, with defining occurrences in the present program, may have applied occurrences elsewhere. Any required information is supplied immediately preceding the program.

This information takes three forms. The first form consists of any mode-declarations which might be necessary for the other two forms. These mode-declarations are to be treated by the compiler as if they occurred in range number zero (i.e., outside the outermost block) of the program.

One other form, called an 'externation', is a list of all the identifiers which have their defining occurrences external to the program being compiled and have applied occurrences within it. The syntax (in terms of the syntax in [6]) for externations is:

```
externation: external symbol, item list.
item: formal parameter, renaming option.
renaming: equals symbol, identifier.
```

A representation for external-symbol is external. The order of the items in the item list is irrelevant. Consider the following example of an externation:

```
external real variablename = outname,
      proc ( real , int ) real name
```

There are two items in this externation. The first item makes use of a renaming (to specify the identifier outname, as described below). The identifiers variablename and name may have applied occurrences within the program being compiled. More detail on externations is given below.

The last form of information is a list of identifiers which have their defining occurrence in range number one of the program being compiled and which may have applied occurrences within independently compiled programs. This list is called a 'globalation', for which the syntax is:

globalation: global symbol, gitem list.

gitem: identifier, renaming option; virtual plan, operator, renaming.

A representation for global-symbol is global. The order of the gitems in the gitem list has no significance. Consider the following example of a globalation:

```
global insidename = outname,  
    ( real , int ) bool + = ribplus, name
```

There are three gitems in this globalation. The first makes use of renaming to specify an identifier (outname) which can be used in an externation to refer to the identifier insidename. The second gitem must contain a renaming since an operator is involved and only identifiers may be used in externations. The identifiers insidename and name as well as the specified operator + must all be defined within the program being compiled. More detail on globalations is given below.

4.2 Facilities Provided

The above-mentioned additions will now be considered with regard to the facilities they provide to programmers. Also, details of the uses that the compiler has for the information provided by means of these additions, are described.

An externation is the means by which a programmer informs the compiler of all the identifiers (and their modes) which have applied occurrences within the program being compiled, but have their defining occurrence in an independently compiled program. The compiler is to assume that these programs will be executed together and, therefore, allotting of storage, for the value of any of these identifiers, is done by the program which contains the defining occurrence of the identifier.

The identifier contained in the formal parameter of an item is the identifier which can have applied occurrences within the program. In order to allow programmers the ability to use meaningful identifiers within any one compilation and still not limit the length of identifiers to eight characters (as is required by many system programs), the renaming option is included in the syntax. This allows an identifier defined in one program to have applied occurrences in other programs, under the alias specified by

the renaming. The renaming is used to specify which identifier, if different from the one of the formal parameter, is 'made global' (as described below). The identifier of the renaming (if there is one) will be referred to as an 'external identifier' (or, an identifier 'made external'). If there is no renaming, then the identifier of the formal parameter will be the one made external. Note that applied occurrences, in the program, of the identifier of a formal parameter correspond to a global identifier which is identical to the external identifier of the item containing the formal parameter.

By this method, the compiler is provided with the information required for handling applied occurrences of identifiers which have their defining occurrence in a different program. The compiler uses this information for producing an external reference (ESD record) for each external identifier. The compiler also produces an 'external table' which contains all the external identifiers and their associated mode numbers. This facilitates checking, at linkage time, whether the global identifier and identical external identifier (from a different program) are, in fact, of the same mode (as is necessary for proper communication).

A globalation is the means by which a programmer informs the compiler of any identifiers or operators which have their defining occurrence within range number one of

the program, and which may have applied occurrences in independently compiled programs. The defining occurrence must occur in range number one to ensure that an applied occurrence of an identifier has a defining occurrence to identify. The identifier which is considered 'global' (or 'made global') is the identifier of the renaming, if any. If there is no renaming, the identifier which is the gitem is considered global. It is the first identifier of a gitem which has its defining occurrence within the program. The renaming is only included in the syntax so that identifiers used in programs are not restricted in any manner. It is the identifier made global which can be used in an externation as an external identifier.

The compiler uses the information about the global identifiers to produce an ESD record for each global identifier. These records specify, for the linking programs, the locations of the values of the identifiers. The compiler also produces a 'global table' which contains all the global identifiers and their associated mode numbers to facilitate checking at linkage time whether the global identifiers and identical external identifiers (from different programs) are in fact of the same mode.

The globalation and externation for a main program and procedure will now be considered. For a main program, the externation contains an item for each independently

compiled procedure with which it will be executed. The globalation contains a gitem for each identifier (or operator) which has its defining occurrence in range number one, and which may have applied occurrences within the procedures which are to be linked with the main program.

For a procedure, the externation contains an item for each identifier which has an applied occurrence within the procedure, but has its defining occurrence in an independently compiled main program. The globalation contains only a gitem for the global identifier referring to the procedure itself (note that this could be a default).

Global (and external) identifiers, other than those referring to a procedure, must have their defining occurrence within a main program. This is to ensure that an applied occurrence of an identifier can identify the defining occurrence. Note that identifiers defined within a procedure can only be identified (by an applied occurrence) during the duration of the execution of the procedure.

It is obvious that use of these additions to supply the compiler with the necessary information has put no restrictions on the order in which main programs or procedures need to be compiled. Also, no restrictions exist on the identifiers used in a program, because the communication between programs is achieved using the external and global identifiers, which are not necessarily

the same as the identifiers used in the programs. Thus the above method allows the execution of independently compiled programs together, with a high degree of intercommunication and no major restrictions. An alternate method for providing the necessary information to the compiler is described and evaluated in Chapter seven. Solutions to the execution time problems are discussed next.

Chapter 5

New Tasks For The Compiler And Linking Programs

When executing independently compiled ALGOL 68 programs together, one uniform set of modes and mode numbers must, in general, be used by all the programs. This uniform set of modes and mode numbers is created at linkage time. The details of the output from the compiler which facilitate this linking, and the actual linking which is done by the linking programs are described in this chapter. Alternative methods for solving the execution-time requirements (which were described in Chapter three) will be discussed and evaluated in Chapter seven.

5.1 The Compiler

The compiler must provide the necessary information for creating the composite mode list and for replacing the old mode numbers in the code with the new ones. It must also provide the information for checking identical global and external (from different programs) identifiers to see whether they are in fact of the same mode. All this information is placed in a CSECT in the object module produced by the compiler. This CSECT will be referred to as the 'mode CSECT' (its detailed description is given below).

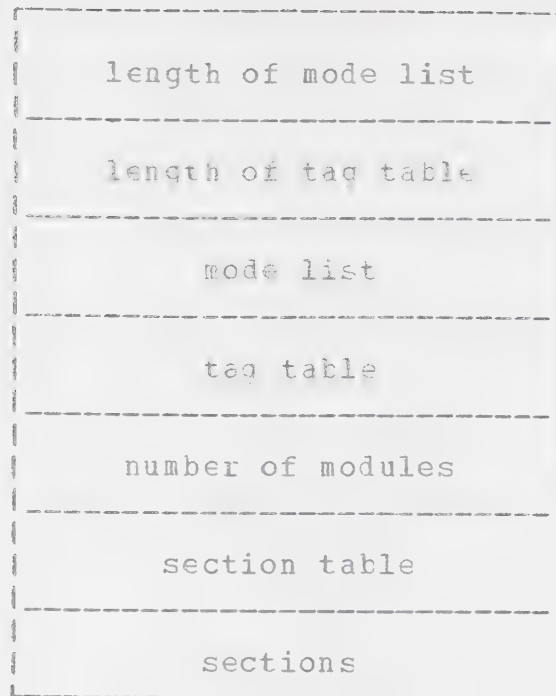


Figure 1

Format of a Mode CSECT

A new ESD type (ESD types are discussed on page 7) is needed to identify mode CSECTs. This ESD type will be called AL (for ALGOL). Whenever a compiler produces an object module for an ALGOL 68 program which will be executed together with other independently compiled ALGOL 68 programs, it must include a mode CSECT and an ESD record for this CSECT. The record will include the associated ESID, the relative starting address of the CSECT, the name of the CSECT (usually all blanks), and the length of the CSECT. Thus the linking programs will be aware of mode CSECTs in modules.

A mode CSECT will now be described in detail (see Figure 1). Note that there is no executable code in a CSECT, but rather lists, tables, counters and working space for the linking programs to use.

The first two parts of the CSECT are the lengths of the mode list and tag table (a table containing all the selectors for struct-type mode descriptions in the mode list). These lengths are included because they may vary between different mode CSECTs.

The next part of the mode CSECT is the mode list (see [9] and [10] for an alternate mode list format). This contains a description of each mode used in the module (or modules, in the case of a library). The mode list consists of an indeterminate number of entries which are of fifteen different types. The different types are:

int, real, bool, char, format, void, bits, bytes, sema,
row, long, ref, proc, union, and struct.

Each type is assigned a 'mode type number'. These numbers are consistent between different mode CSECTs. Possible mode type numbers for the above types might be zero through fourteen. These numbers have no relation to mode numbers, which are used to associate identifiers to their modes. The format of the different types of entries will now be given. The fields of the entries are delimited by the character | in the formats of the various table (or list) entries given

in this chapter.

The mode list contains only one entry for each of: int, real, bool, char, format, void, bits, bytes, and sema. These entries form the constant portion of the mode list and are the same in every CSECT. The entries for these types are of the following format (the zero is included for address alignment purposes on the IBM 360 series of computers; there is no other significance to this field).

```
| mode type number | zero |
```

The other entries vary in format, depending on their mode type. Where 'displacement' is used in the format of a type, it is the displacement from the beginning of the mode list where a mode description, which is being referenced, begins. The abbreviation 'm.t.n' will be used to mean 'mode type number'.

Entries of types row (used for arrays) and long are of the following format:

```
| m.t.n | number | displacement |
```

For entries of type row, the number in the format is the number of dimensions and the displacement points at the mode description for the elements of the array. For entries of type long, the number in the format is the number of times long is used in the mode-declarer, and the displacement points at the mode description being referenced.

Entries of type `ref` are of the following format (the zero is for alignment purposes again, and has no other significance):

```
| m.t.n | zero | displacement |
```

The displacement in the above format points at the mode description which is referenced by the ref in the mode declarer.

Entries of type `proc` are of the following format:

```
| m.t.n | number (m) |...| displacement |...|
```

There are `m` displacements in the above format, one for the mode of each parameter, followed by one for the mode of the value (if there is one) returned by the procedure. If the number of displacements in the mode description (prior to being placed in the mode list by the compiler) was not odd then an extra negative-valued displacement is included by the compiler, at the end of the entry (in order to maintain address alignment on the IBM 360, and thus may not apply to other implementations) and `m` is increased by one. This negative-valued displacement is ignored when considering the mode description in the mode list.

Entries of type `union` are of the following format:

```
| m.t.n | number (m) | length |...| displacement |...|
```

There are `m` displacements in the above format, one for each of the modes used in the union. The length field contains the maximum length which a value of this mode may occupy in

memory. If the number of displacements in the mode description (prior to being placed in the mode list by the compiler) was not even then an additional negative displacement is included by the compiler, at the end of the entry (for alignment purposes on the IBM 360) and m is increased by one. This negative-valued displacement is ignored when considering the mode description in the mode list.

Entries of type struct are of the following format:

```
| m.t.n | number (m) | length |...
```

```
| disp | displacement |...|
```

There are $m/2$ pairs of the form | disp | displacement | in the above format. 'Disp' is the displacement from the beginning of the tag table where the tag, corresponding to the field of the struct whose mode description is pointed at by the displacement in the pair, begins. Tags are the (programmer) assigned selectors of the fields of a structure. There is one pair (as described above), in a mode description, for each field of a structure. 'Length', is the length in memory which the non-varying portion of a value of this mode would occupy.

Next in the mode CSECT is the tag table. This table contains all the tags for the entries of type struct in the mode list. The entries in the tag table are of the following form:


```
| length | identifier |
```

The length is the number of characters (minus one) in the identifier. If necessary, the tag table is padded at the end to align (again an IBM 360 dependency) the next part of the mode CSECT.

Let us now consider a sample mode-declarer and its description in a mode list. Consider the mode-declarer:

```
proc ( struct ( real first, bool second), long long int )  
  ref int
```

Assume that the mode type numbers zero through six were assigned to the types int, real, bool, long, ref, struct, and proc respectively. All displacements in the mode list (in Figure 2) are labelled beginning with the letter m (note that these are the mode numbers). Displacements in the tag table are labelled beginning with the letter t. The type of entries that would exist in a mode list and tag table are in Figure 2. The displacements are indicated on the left of the entries. The mode declarers corresponding to the entries are: int at ma, real at mb, bool at mc, long long int at md, ref int at me, struct (real first, bool second) at mf, and the whole mode declarer (proc etc.) at mg.

Mode List

ma	zero	zero					
mb	one	zero					
mc	two	zero					
md	three	two	ma				
me	four	ma					
mf	five	four	length	ta	mb	tb	mc
mg	six	three	mf	md	me		

Tag Table

ta	four	first
tb	five	second

Figure 2

Sample Mode List and Tag Table

The next field is the number of modules. This is one unless this CSECT is for a library, in which case the number is equal to the number of ALGOL 68 modules in the library which reference the mode CSECT. There is one entry in the

section table for each module and there is also one section for each module.

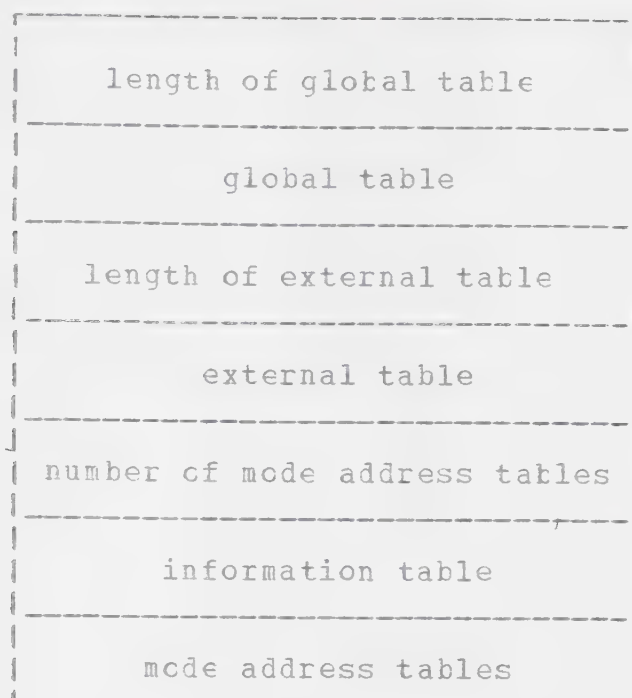


Figure 3

Format of a Section

Each section table entry consists of two fields. The first field is zero unless this is a mode CSECT for a library, in which case the field is a pointer to an ALGOL 68 module in the library. The second field contains the length of the corresponding section. A section contains the information pertinent to one module, such as its external and global tables and also a description of where mode numbers exist in the code of the module. Thus, sections provide information which allows for checking communication (that is, checking mode numbers of corresponding global and

external identifiers) and replacing old mode numbers by the appropriate new ones. The format of a section (see Figure 3) will now be considered.

The lengths of the global and external tables are included because the size of these tables may vary between different CSECTs . The global table has one entry for each global identifier in the module. The external table has one entry for each external identifier in the module. The format of entries in both tables is:

```
| identifier | feature flags | mode number |
```

The feature flags indicate certain options, under which the module containing the external or global identifier was compiled. Identical identifiers from modules being executed together, must have their feature flags match, thus indicating that the code in the modules is compatible. Examples of such options are initializing values (that is, the identifier expects its value to be initialized) and scope checking (that is, a check is made to determine if the value of the identifier exists in the blocks of code where it is referenced). There are two flags for each option. The first flag indicates whether the option is relevant for the particular identifier. If so, then the second flag indicates whether the option should be on or off. In order to have proper communication, feature flags must match for identical global and external (from different

programs) identifiers.

The number of mode address tables is equal to the number of ESD records used to define portions of code containing mode numbers. The information table has one entry for each mode address table. Each entry has three fields. The first field is used by the linking programs and is left blank by the compiler. The second field has the length of the corresponding mode address table. The third field has the ESID (identifying the ESD record) with which the corresponding mode address table is associated.

The mode address table entries are of two forms. In either case, the value of the entry indicates a place in the code where a mode number exists. If the entry is positive, the referenced mode number is to be replaced by the appropriate new mode number whenever this module is combined with others. If the entry is negative, the referenced mode number is to be replaced by the actual memory address of the mode description to which it refers, when the program is loaded; under other circumstances, such as in library building, it is replaced by the appropriate mode number. Thus, mode address tables inform the linking programs of the locations of mode numbers in the code.

The description of the mode CSECT is now complete. The actual linking (with respect to ALGOL 68 modules) which is done by the linking programs will now be described.

5.2 The Linking Programs

Forming a composite mode list and replacing the old mode numbers by the appropriate new mode numbers can be done by any of the linking programs.

A library building program produces a library from an input of object modules, load modules or other libraries. These input modules can be ALGOL 68 modules or otherwise. There is no restriction as to the types of modules in a library. A library builder is able to form a composite mode CSECT and replace the old mode numbers, by the appropriate new ones, in all the ALGOL 68 modules which had a mode CSECT on input. The composite mode CSECT is the last entry in the library and can be accessed via the last entry in the directory. The mode CSECTs from the various input modules are not put into the library because all the modules in the library use the same composite mode CSECT. However, the ESD records in the modules are left intact, for they signify that the module references the mode CSECT of the library.

There are several reasons why a library builder does this linking. The first is that having one mode CSECT rather than several saves a significant amount of space in the library. The reason for this saving is that the mode list in the composite mode CSECT has only one mode description for

any two or more equivalent mode descriptions which existed in the various input mode CSECTs. Since linking must be done (by one of the linking programs) between modules that will be executed together, another reason for a library builder to do this linking is to save time and space when loading several modules from a library. Time is saved in that the loader is not required to spend the time linking these modules. Space is saved in that less working storage is required by the loader to build a composite mode list. There can be a substantial saving when loading several ALGOL 68 modules from one library and hence the choice of modules to be put into one library should be made with this in mind.

A library can have one mode CSECT for several modules since a mode CSECT can contain the information for an arbitrary number of modules. This is so because the number of modules (field) can be changed; the section table can be increased and sections can be added on to the CSECT. There is essentially no limit to the number of modules to which a mode CSECT can apply, using this design.

The library builder forms the composite mode CSECT in the following manner. If there is only one input mode CSECT, then it is used and nothing else is done in this respect. If there are several input mode CSECTs, then a composite mode CSECT is formed. First, a composite tag table is built by combining all the tag tables from the input

CSECTs. The composite mode list is then formed (as described in Chapter six). The composite mode list, its length, the composite tag table and its length are then put into the composite mode CSECT. The mode numbers in the global and external tables of the input CSECTs are then replaced by the appropriate new ones. Then the number of ALGOL 68 modules to which the composite mode CSECT applies is put into the CSECT. The section table (a composite of the input section tables) and the sections are then put into the composite mode CSECT. Once this is complete the library builder forms the rest of the library. While it is doing this it replaces the old mode numbers, in the code of the appropriate ALGOL 68 modules, by the proper new mode numbers. It also puts the pointers to the appropriate modules into the section table. When it has finished building the rest of the library, it puts the mode CSECT at the end of the library and it puts an entry for the CSECT at the end of the directory. The library is then complete.

The linkage editor must also be able to do this linking. A load module must appear just like an object module. Thus, it must contain only one mode CSECT even though it may have been composed of several ALGOL 68 modules which contained mode CSECTs. It is possible for one mode CSECT to be used for a load module because the mode address tables are related to the module via the ESIDs in the information table. Thus, when the load module is created,

all the old ESIDs in the input mode CSECTs must be replaced by the appropriate new ESIDs. Once this is done, the linkage editor can build the composite mode CSECT as did the library builder, except that, for a load module, the section in the composite mode CSECT is a concatenation of the input sections with their ESIDs properly renumbered, whereas in a library, the individual sections are put into the composite mode CSECT because the individual modules still exist. The feature flags and mode numbers are tested between identical global and external identifiers and if they do not match then the load module is not produced, for the module will not be able to execute properly. If they do match then the old mode numbers in the code of the input ALGOL 68 modules are replaced by the appropriate new ones and the composite mode CSECT is put into the final load module.

The loader is also able to do this linking. It forms a list of the mode CSECTs as it encounters them. While doing this it fills in the blank word in each information table entry with the corresponding pointer to the ESD table. Thus information such as an ESID's appropriate memory address and relocation factor are accessible via the mode CSECTs when the composite mode list is built. The loader keeps track of those library modules which have been loaded and which reference the library's mode CSECT, by putting a zero where the file pointer used to be in the section table of the

library's mode CSECT. Note that a library's mode CSECT is not put into the list of mode CSECTs unless a module from the library is loaded which has an AL type ESD record.

Once this mode CSECT list is completed and all the necessary modules are loaded, then the loader is ready to form the composite tag table and mode list. It does not form a complete composite mode CSECT (as did the library builder and linkage editor). This is because the rest of the CSECT's fields are used only for replacing mode numbers in the code and these fields are no longer necessary after the program is loaded. The loader builds the composite tag table and mode list as did the other linking programs. The mode numbers in the global and external tables of all the input CSECTs must then be replaced by the proper new ones. Then the loader must check each global and identical external identifier to see if they are of the same mode and to see if their feature flags match. If an error is detected, the programs cannot execute properly so the loading process is terminated. If no errors are found, the loader replaces the old mode numbers in the appropriate ALGCL 68 modules with the proper new ones. It replaces the mode numbers referenced by a negative entry in the mode address table by the memory address of the appropriate mode description. The loader then releases the space occupied by the list of mode CSECTs, for they are no longer needed. Only the composite mode list and tag table are loaded with the programs. The loading process

is then finished.

This completes the description of how the composite mode list and uniform set of mode numbers is achieved by execution time. While doing this linking, it is possible to check the communication between the linked programs, which is a desirable facility. One thing which considerably simplifies the processes involved in combining modules is the generality of the mode CSECT. The format of the CSECT is such that it can be used for an object module, load module, or library . Thus, in the formation of composite mode CSECTs, no special case need be made regarding the input mode CSECTs. Several different methods for satisfying the execution time requirements will be described and evaluated in Chapter seven. A detailed description of the algorithms used by the above linking programs in forming the composite mode list and replacing mode numbers will now be discussed.

Chapter 6

Algorithms

The linking programs have been assigned several jobs in order to satisfy the execution time requirements which have been described. The algorithms which the linking programs use for the jobs of forming a composite mode CSECT and replacing the mode numbers in the code by the appropriate new mode numbers will now be described in detail.

6.1 Equivalence Of Mode Declarers

The linking programs must form a composite mode list from the input mode lists. This composite mode list must consist of only non-equivalent mode descriptions. This means that a method for determining equivalent mode descriptions is required. To date, basically two methods have emerged. The first method was first described by C.H.A. Koster (see [11]) and the other by M. Zosel (see [10]). These two methods will now be discussed. Also, details of the version of Zosel's method which is used here by the linking programs will be given.

The algorithm suggested by Koster for determining

the equivalence of declarers is based on the principle that the 'full expansion' of equivalent declarers will be the same. A declarer is 'expanded' by replacing any mode-indications in the declarer with their defining declarer. The full expansion of a declarer is obtained by expanding it until no mode-indications remain. However, the full expansion of an infinite mode (such as: mode m = struct (ref m name)) is infinite, and thus cannot be achieved. To determine if two infinite modes are equivalent their declarers must be expanded until it is obvious whether they specify the same mode. Expanding a declarer does not change the mode specified by it. Consider the declarations:

mode x = ref y, y = int , z = ref int

The defining declarer of y is int . The full expansion of x is ref int . Thus x is equivalent to z , as can be determined by comparing the full expansions of x and z element-by-element. Therefore, the structure of the mode list should be such that this comparison can be done easily.

Koster's method is useful in an implementation in which the equivalence of two declarers is checked only when necessary, i.e. for conformity relations. This method is not well suited for a system where all the declarers are compared and equivalent descriptions are removed from the mode list. A method which is more appropriate for such a system was suggested by Zosel. Koster's method describes the basic procedure for determining whether two declarers are

equivalent, and thus leads the way to more powerful methods.

Zosel introduced a method which is very good for systems where all (duplicate) equivalent mode descriptions are removed from the mode list. This method uses the insights of Koster's method as well as an algorithm similar to one for minimizing finite state machines. Mode list entries similar to those described in Chapter five are used. The algorithm will now be described.

The first task is the division of the mode list entries into classes according to their mode type. Then, each class is subdivided such that the only mode descriptions remaining in any given class have all their displacements (fields) in identical classes. When no more subclasses can be formed in this manner then each of the remaining classes is an equivalence class. If any class contains more than one mode description, at this point, then those descriptions are equivalent. The power of this method is in the fact that the class structure avoids the necessity of a complete element-by-element comparison of expansions of two declarers to determine if they are equivalent. These expansions are essentially tree structures and the class structure, as described, allows comparing of whole trees (or parts of trees) at once by looking at the class number. The element-by-element comparison is actually done once when the various classes are formed.

We will now consider an example of this method.

Assume the following mode-declarations:

```
mode x = ref proc y ;
mode y = [1:n] int ;
mode z = ref xx ;
mode xx = proc [ ] real ;
mode yy = ref proc [ ] int ;
```

The normal forms (according to Zosel) for the above modes, when divided into classes, are:

```
class one:  x = ref m1
            z = ref xx
            yy = ref m2

class two:  m1 = proc y
            xx = proc m3
            m2 = proc m4

class three:  y = row m5
              m3 = row m6
              m4 = row m7

class four:  m5 = int
              m7 = int

class five:  m6 = real
```

The classes are then checked for non-equivalent mode descriptions (i.e. those with corresponding fields in different classes). The first class that can be subdivided is class three because m6 is in a different class than m5

and m7. Class two can then be subdivided, because m3 is in a different class than y and m4. Then class one can be subdivided, because xx is in a different class than m1 and m2. The classes then are:

```

class one:   x = ref m1
              yy = ref m2
class two:   z = ref xx
class three: m1 =proc y
              m2 = proc m4
class four:  xx =proc m3
class five:  y = rcw m5
              m4 = rcw m7
class six:   m3 =row m6
class seven: m5 = int
              m7 = int
class eight: m6 = real

```

The classes cannot be subdivided further because, in each class, the fields in the mode descriptions are in identical classes. Thus x and yy are equivalent, as are m1 and m2, y and m4, and m5 and m7.

This method has several complicated aspects, such as the method used for recording and forming classes. Also, the description given presumes that the method will be used by the compiler, and for only one mode list. There is no mention of the problems involved when several mode lists are to be merged into one composite mode list, as must be done

by the linking programs. Details of the implementation, within the linking programs, of a version of Zosel's algorithm will now be given.

All the linking programs must be able to form a composite mode list. In preparation for this, a list is made of the mode CSECTs which are to be merged so they can be easily accessed. The composite mode list is then filled in with all the entries from the old mode lists. However, a problem which occurs is that the displacements in the mode descriptions of the entries put into the composite mode list are relative to different addresses. Hence, when a mode description is put into the composite mode list then the displacements in the description must be altered to be relative to the beginning of the composite mode list. However, it cannot be guaranteed that a given mode description will be in the composite list at the time when a mode description referencing it is being entered, and therefore the displacements cannot be updated until all the entries from the input mode lists have been entered.

To facilitate this updating, each mode description in the input mode lists is replaced by the displacement at which it is entered into the composite mode list. Also a table (work mode table), of the same size as the initial composite mode list, is used. When a mode description is put into the composite mode list, the address of the input mode

list (from which the mode description was taken) is placed in the work mode table at the same displacement.

The composite tag table must also be built before this renumbering in order to be able to renumber the disps (displacements in the tag table where a tag begins) in the struct mode type entries. The composite tag table is the serial composition of all the old tag tables. The displacement of each old tag table from the beginning of the composite tag table is kept in another table to simplify renumbering the disps.

When the composite mode list and tag table have been formed, the mode descriptions in the composite mode list can be updated, using the work mode table and the input mode lists. The address of the mode list from which the mode description came is added to each displacement. This address was in the work mode table at the same displacement as the mode description in the composite mode list. This sum is the original position of the mode description (which the displacement references) in the input mode list. However that mode description was replaced by the displacement of its new position in the composite mode list. This displacement is, in fact, the appropriate new displacement which replaces the displacement we began with. Thus the old displacements can easily be replaced by the appropriate new ones.

In the case of struct type entries, the disps are added to the displacement at which the tag table to which they refer was put into the composite tag table. When this updating is complete, the checking for equivalent mode descriptions can be done.

The initial class table is built in preparation for the checking of equivalent mode descriptions in the composite mode list. This building is done while the composite mode list is formed. The initial class table has one class for each mode type. The entries in the class table are of the format:

```
| number | flag |...| displacement |...|
```

The 'number' field contains the number of entries (displacements) in the class. The flag specifies whether the class is an equivalence class (cannot be further subdivided). The displacements refer to the mode descriptions, in the composite mode list, which are in this class.

After the entries in the composite mode list have been updated, the entries in the work mode table are replaced by the class number (the address of the beginning of the class) of the corresponding mode description. This is done to speed up the checking of class numbers later in the algorithm.

Checking for equivalent mode descriptions is done in the following manner. Each non-equivalence class (one which is not flagged) is checked to see if it can be subdivided. The only mode descriptions that are to remain in a class are those which are equivalent to the mode description pointed at by the first entry (displacement) in the class. The class is considered an equivalence class (and so flagged) if all the displacements in the mode description pointed at by the first displacement in the class, are themselves in equivalence classes. A new class is formed at the end of the class table of all the other displacements in the class being checked. When a new class is formed, the class number in the work mode table must be changed appropriately for each element of the new class. If a pass is made through the class table without any new classes being formed, then each of the classes is an equivalence class, and this portion of the algorithm is finished.

Using the above method it can be determined whether two mode descriptions are equivalent by checking whether the corresponding displacements in the two descriptions belong to the same class. This is easily done, as the work mode table contains the class number of the mode descriptions at the displacements under investigation.

The final composite mode list must not contain any equivalent mode descriptions. Thus, the next job to be done

is the removal of all but one mode description from each equivalence class. As this is done, the work mode table entries are altered to reflect the changes being made. The removal of redundant mode descriptions is effected by using two pointers, a back displacement and a forward displacement. The back displacement is the displacement from the beginning of the composite mode list at which the next mode description is to be put. The forward displacement is the displacement at which the current mode description, which is being checked, begins. Both forward and back displacements are initially set to zero. Each mode description is checked to see whether there is already an equivalent mode description in the composite mode list by checking the first field of the class to which the mode description belongs. If the field contains a number greater than zero then no equivalent mode description has been put into the final composite mode list and so the current mode description is moved to the location specified by the back displacement. The value of the back displacement is put into the work mode table at the forward displacement. The class entry is changed, to a zero in the first field and to the back displacement in the second field. Then the back and forward displacement are incremented by the length of the current mode description, and the next mode is checked. If the first field of the class to which the mode description belongs is zero then the second field (which contains the

displacement of the equivalent mode description already in the final mode list) is copied into the work mode table starting at the forward displacement. The forward displacement is then incremented by the length of the current mode description. This process continues until the forward displacement becomes the length of the initial composite mode list. At this point there is one mode description in the composite mode list for each of the equivalence classes. The final length of the composite list is the value of the back displacement.

There is one more necessary job in forming the final composite mode list. All the displacements in the mode descriptions must again be updated so as to point at the appropriate mode descriptions in the composite mode list. The deleted mode descriptions and moved mode descriptions cause this inaccuracy in the displacements. This updating is done by using the work mode table which contains the appropriate new displacement in the composite mode list for each old displacement at the old displacement in the work table. So, each displacement in the mode descriptions is replaced by the entry in the work mode table which is found at that displacement. The composite mode list is then complete and accurate.

6.2 Specifics Of The Linking Programs

After forming the composite mode list, the various linking programs have slightly different jobs. These jobs will now be described for the individual programs.

The library builder must build a complete composite mode CSECT after forming the composite mode list. The displacements in the various global and external tables in the input mode CSECTs are then updated. This is done by replacing them with the entry in the work mode table found at the displacement which is to be updated. The library builder can then form the complete mode CSECT. It determines the total number of modules and, thus, the length of the section table. It then completes the composite mode CSECT by filling in the section table and sections serially from the input mode CSECTs.

The only job remaining for the library builder is the updating of the mode numbers in the code of the various modules. This is done in the second pass made through the input modules. During the first pass, the library builder determines the size of the directory, forms the list of input CSECTs and then forms the composite mode CSECT. It is during the second pass that the library is actually formed. As each ALGOL 68 module is put into the library, its old mode CSECT's mode address tables are used to indicate the

location of the mode numbers in the text of the module. Each such mode number is replaced just as the other displacements (in the composite mode list , external and global tables) were replaced. That is, the mode numbers (displacements) are replaced by the entry in the work mode table starting at that displacement which is being replaced. Each module is put into the library after its mode numbers have been replaced. The old mode CSECTs are not included in the library as the modules use the one composite CSECT for the library. However, the AL ESD records are left in the modules to indicate that the modules require the composite mode CSECT. The composite mode CSECT is put into the library at the end and an entry for it is put into the directory.

The linkage editor must also build a complete composite mode CSECT. This is done in essentially the same manner as by the library builder. First, the rest of the load module is formed. The ESD records for the mode CSECTs, as well as those CSECTs themselves, in the various modules are not included in the load module. A complication which is unique to the linkage editor is due to the fact that the ESIDs are changed in the various ESD records as they are put into the load module. Thus, as the load module is being built, a table (ESID relation table) is kept which relates the old ESIDs to the new ones and also contains the address of the code referred to by the ESIDs. Once the rest of the load module is complete, an ESD record with the proper ESID

is included for the composite mode CSECT. The composite mode CSECT is then formed.

At this point the identical global and external identifiers are checked to see whether their mode numbers and feature flags match. If they do not, then the linking is terminated with an error message.

Since the sections in the composite mode CSECT are just the sections from the input CSECTs, they contain the out-of-date ESIDs, which must now be updated. This is done using the ESID relation table. When this is accomplished, the mode numbers in the code are updated, in the same fashion as during library building except that the ESID relation table is used to specify the location in the code referred to by an ESID.

The composite mode CSECT is then placed into the load module and the load module is complete. It is obvious that, except for some additional code to handle the change in ESIDs, essentially the same code can be used for both the library builder and the linkage editor.

The loader's functions are slightly different from the other two programs. It must first load all the modules which are to be executed together. If it loads a module from a library which contains an AL ESD record then it checks whether the library's mode CSECT is already loaded. If it is

not, then the loader loads it, using the fact that the mode CSECT is the last module in the library and that it is accessible via the last entry in the directory. When the mode CSECT is loaded then the file pointer (in the section table) to the loaded module, which references the CSECT, is changed to a zero. This way the loader is able to determine which of the sections to use later when updating mode numbers in the code. As each module is loaded (from a library or otherwise), the first field of each information table entry in the appropriate section is set to the address at which the code for the ESID in the third field of that information table entry was loaded. Also, as the modules are loaded, a list of the input mode CSECTs is made. When the loading of modules is complete, the composite mode list is formed in the same manner as for the other linking programs.

The displacements in the external and global tables are then replaced by the appropriate new ones (as before). Then the corresponding mode numbers and feature flags of identical external and global identifiers are compared. If they do not match, then loading is terminated.

The mode numbers in the code are then updated to be consistent with the composite mode list. This is slightly different from the other two programs, for the loader recognizes (see mode address table entries in 5.1) the difference between the mode numbers which are replaced by

displacements (as done by the other programs) and those mode numbers which are to be replaced by the absolute address of the mode description. Placing absolute addresses in the code allows the program to access the mode list during execution, thus satisfying the needs outlined in Chapter three. When mode numbers are replaced by appropriate mode numbers, this is done in the same manner as in the other programs. When an address is required (as specified in the mode address table) the loader adds the address of the beginning of the mode list to the proper displacement and puts the sum into the specified location. When this is completed the loading process is finished. Only the composite mode list and tag table are formed; the rest of the mode CSECT is not needed at execution time and is therefore not built.

This completes the description of the algorithms used in the linking programs. It is important to realize how the generality of the design of the mode CSECT has allowed essentially the same techniques and code to be used in the different programs. It is important that the format of a mode CSECT allow it to be used for either a load module, object module or library. The designs of the mode list, work mode table and class table were made to facilitate the execution of a version of Zosel's algorithm for determining (and removing) equivalent mode descriptions in a mode list.

Chapter 7

Alternative Methods

There are several other methods for satisfying the compilation and execution time requirements which exist when executing together independently compiled ALGOL 68 programs. These methods will now be described and evaluated.

The implementation of ALGOL 68R (see [12]) includes a method for satisfying both the compilation and execution time requirements at compilation time. This method involves the use of two lists. One of these, the 'KEEP' list, contains all the identifiers, operators and modes which have their defining occurrence in the program being compiled and may have applied occurrences in programs which are to be compiled. The order of the elements in the list is immaterial. This list is included at the end of the program being compiled. The segment (object module) which is produced during this compilation contains the mode list used during the compilation, and the elements of the KEEP list and their corresponding mode numbers. After the compilation, the segment may be put into an album (library) or it may be executed.

The other list is the 'WITH' list; it contains the

names of all the previously compiled segments which are to be placed into the segment formed during the current compilation. This list is included immediately in front of the program being compiled. The order of the names in the list is the order in which the referenced segments will be placed, by the compiler, into the segment produced. The segments named in the list are loaded from the album which is specified at the end of the WITH list by 'FROM filename'. This loading is done prior to compilation of the program which follows the WITH list. The mode lists in the loaded segments form the initial mode list for the current compilation. The names in the KEEP lists of these segments may have applied occurrences within the program being compiled. Note that any segment which is necessary for the proper compilation of the program must already have been compiled and must be named in the WITH list.

This method has several good features. First, it allows a high degree of communication between independently compiled segments, since identifiers, operators and modes can be on a KEEP list. Also, very little extra (with regard to the program being compiled) information is actually supplied by a programmer, for the elements of the two lists are just names.

This method, however, has several very poor features. The most obvious one is the requirement that all

the independently compiled programs which are necessary for a particular compilation must have been compiled previously. This is very restrictive when a large program is divided into sections. A section of the program which depends on another cannot be compiled until that necessary other section has been compiled. This hampers parallel work on the different sections of the program.

The communication between the various sections is restricted by the order in which they are to appear in the final segment. In any section there cannot be applied occurrences of names which are in the KEEP list of a section which is placed after it in the segment. Consider a program divided into sections A, B and C (the order in the final segment being A then B then C). Section A must be compiled before B and C if B and C contain applied occurrences of names on the KEEP list in A. A cannot have applied occurrences of names on the KEEP lists of B and C. Similarly, B must be compiled before C. Also, B cannot have applied occurrences of names used in C. This method is very wasteful, for, in order to compile C, both A and B must be loaded first. Thus when C is being written and tested there is a large overhead at each test compilation due to this pre-compilation linking. This overhead can be very costly if there are many sections in a program.

There is one more weakness in this method. The

communication between the various independently compiled sections of a program is not checked at all. The case where different sections of the program have applied occurrences of the same identifier, but are assumed to be of different modes for that identifier, may not be discovered at compilation or execution time. This situation may result in an error that is difficult for a programmer to detect. All the above situations exemplify how restrictive and wasteful this method can be. The poor features of this method are felt to far outweigh the good features.

There are two other methods that might be considered for fulfilling the execution time requirements. These methods could be used in conjunction with the method described in Chapter four for fulfilling the compilation time requirements. Both these methods require the independently compiled modules to contain the mode list formed when the module was compiled.

One method consists of forming a single mode list, at execution time, from the mode lists of the various input modules which are being executed together. This formation could be done as described for the linking programs. All the mode numbers in the code could then be updated and then execution could begin. A good aspect of this method is that the standard linking programs would not need to be changed at all to accomodate ALGOL 68 programs.

However, there are several poor aspects to this method. First, execution time is increased. Thus if a program were to be executed many times, there would be a large extra expense involved because of this 'mode work', which would be done every time the program was executed. Also the loader and linkage editor could no longer check the communication between the independently compiled modules, for the mode numbers would not be consistent until execution time. Thus, checking could not be done until execution time. Another consideration is that the amount of storage needed would be increased due to the code necessary for doing this mode work. This code would need to be included with every main program. All these detriments, the most important being the increased time and storage needed at execution time, result in this not being a good approach.

Another method for fulfilling the execution time requirements would require the mode lists from all the modules which are being executed together to remain in core for the duration of execution. The equivalence of two mode descriptions could be checked using Koster's algorithm. This would be done only when necessary, i.e. for conformity relations. This method has the advantage that no changes would be required of the standard linking programs, and it does not require the updating of the mode numbers in the code.

However, this method has some poor features. The amount of storage required at execution time would be considerably larger if executing together several modules, each containing its own mode list. Also, storage would be required for the code needed to do the checking of equivalences between modes. The size of libraries would also be considerably increased using this method. There is no guarantee that equivalence checks would be necessary only a small number of times. If the number is large then execution time may be significantly increased. If the execution time requirements are fulfilled in this way then a method is required for associating any one mode number to the proper mode list so that the garbage collector (as discussed in 3.3) can function properly. Also the checking of communication between modules cannot be done until execution time. Therefore, as with the other methods discussed, this does not seem a good approach.

The above methods were all considered as possible solutions to the problems involved in executing together independently compiled ALGOL 68 programs. However, they were rejected for the reasons given above in favour of the method described in Chapters four, five and six. It appears that the best approach to solving the execution time requirements is to place the burden of the mode work on the linking programs.

Chapter 8

Summation

This paper has dealt with the problem of combining independently-compiled ALGOL 68 programs. A general explanation of the three linking programs (the library builder, the linkage editor and the loader) and of those tasks which are commonly performed by these programs was included in order to place in perspective those non-standard tasks assigned to these programs in order to satisfy the requirements of ALGOL 68. The peculiarities of ALGOL 68 which necessitate non-standard functions when combining programs have been given.

Additions to ALGOL 68 are suggested (in Chapter four), usage of which allow programmers to provide the compiler with the information necessary for fulfilling the compilation time requirements. These additions pose no important restrictions on programmers.

The execution time requirements have been solved by use of mode CSECTs and by the performance of non-standard tasks by the linking programs. The format of a mode CSECT is such that it can be used for either an object module, load module or library. This generality modularizes the tasks

involved in combining programs. An important feature of this method is the checking of communication between programs being combined. This can be a very useful debugging aid.

The burden of the mode work has been put on the linking programs. This method was chosen instead of the alternative methods which were described, because it involves the smallest amount of extra storage, and results in no increase in execution time. Thus, a programmer is able to execute together independently compiled ALGOL 68 programs with little extra work and few restrictions.

REFERENCES

- [1] Barclay, J.N., "Linkage-Loader Design in Computer Operating Systems", Masters Thesis, University of North Carolina, Chapel Hill, North Carolina, 1968.
- [2] "System/360 Operating System Linkage Editor", International Business Machines, Form Y28-6667-0, January, 1968.
- [3] "System/360 Operating System Linkage Editor", International Business Machines, Form C28-6538-6, November, 1968.
- [4] "Program Loader", University of Alberta Computing Center, May, 1970.
- [5] Barron, D.W., Assemblers and Loaders , Macdonald/American Elsevier, 1969.
- [6] Van Wijngaarden, A. (editor), Mailloux, B.J., Peck, J.E.I., and Koster, C.H.A., Report on the Algorithmic Language ALGOL 68 , Report MR101, Mathematisch Centrum, Amsterdam, 1969.
- [7] Schorr, H., and Waite, W.M., "An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures", in Communications of the ACM , Volume 10, Number 8, August, 1967, page 501.

- [8] Fites, P., "Storage Organization and Garbage Collection in ALGOL 68", in Proceedings of an Informal Conference on ALGOL 68 Implementation , (Department of Computer Science, University of British Columbia, 1969), page 85.
- [9] Peck, J.F.L., "On Storage of Modes and Some Context Conditions", in Proceedings of an Informal Conference on ALGOL 68 Implementation , (Department of Computer Science, University of British Columbia, 1969); page 70.
- [10] Zosel, M.E., "A Formal Grammar for the Representation of Modes and its Application to ALGOL 68", Doctorate Thesis, University of Washington, Seattle, Washington, 1971.
- [11] Koster, C.H.A., "On Infinite Modes", in ALGOL Bulletin 30, February, 1969, page 86.
- [12] Woodward, P.M., and Bond, S.G., "Users' Guide to ALGOL 68-F", Royal Radar Establishment, Malvern, Worcestershire, England, April, 1971.

B30034